1. Khi nào thì xảy ra tranh chấp?

* Race condition xảy ra khi nhiều process hoặc threads cố gắng truy xuất và thay đổi dữ liệu chia sẻ cùng một lúc. Kết quả cuối cùng của các thay đổi này phụ thuộc vào thứ tự thực thi của các process này.
* Trong ví dụ về khi hai process Producer và Consumer cùng truy xuất biến counter (counter++ và counter--), race condition xảy ra khi thứ tự thực thi của các lệnh tăng hoặc giảm counter không được quản lý đồng bộ. Khi Producer tăng counter và Consumer giảm counter cùng một lúc, kết quả có thể không nhất quán.
* Để giải quyết vấn đề này, cần có các cơ chế đồng bộ hóa, như sử dụng locks, semaphores hoặc các phương pháp khác để đảm bảo chỉ có một process được phép thực hiện thao tác trên dữ liệu chia sẻ tại mỗi thời điểm. Cơ chế này sẽ đảm bảo tính nhất quán và tránh được race condition.

1. Vấn đề vùng tranh chấp (critical section) là gì?

* Vùng tranh chấp (hoặc critical section) là phần của code trong một tiến trình mà tiến trình đó thực hiện các thao tác trên dữ liệu được chia sẻ. Vùng tranh chấp có thể là các đoạn code cần quyền truy cập đồng thời đến dữ liệu chia sẻ, chẳng hạn như biến, bảng dữ liệu, file hoặc bất kỳ tài nguyên nào có thể được truy cập bởi nhiều tiến trình.
* Để đảm bảo tính nhất quán và tránh race condition, mỗi tiến trình phải tuân theo một quy trình đặc biệt để truy cập vùng tranh chấp:

while (true) {

entry\_section();

critical section

* exit\_section();

remainder\_section

}

* + Entry Section: Tiến trình yêu cầu quyền truy cập vào vùng tranh chấp thông qua entry section. Nếu vùng tranh chấp đang được thực hiện bởi tiến trình khác, tiến trình hiện tại phải đợi cho đến khi nào không có tiến trình nào khác thực hiện vùng tranh chấp.
  + Critical Section: Sau khi được phép, tiến trình thực hiện các thao tác trong vùng tranh chấp. Đây là phần quan trọng cần quyền truy cập đồng thời đến dữ liệu chia sẻ.
  + Exit Section: Sau khi hoàn thành vùng tranh chấp, tiến trình thông báo cho các tiến trình khác biết rằng nó đã kết thúc và giải phóng vùng tranh chấp để các tiến trình khác có thể tiếp tục.
  + Remainder Section: Phần còn lại của code sau khi tiến trình đã thoát khỏi vùng tranh chấp, nơi các thao tác không liên quan đến dữ liệu chia sẻ được thực hiện.
* Vấn đề vùng tranh chấp chính là cách thiết kế và quản lý các tiến trình để đảm bảo tính nhất quán và tránh xảy ra race condition khi truy cập vào dữ liệu chia sẻ. Quy trình như entry section, critical section, exit section và remainder section giúp đảm bảo rằng các tiến trình không xâm phạm vào vùng tranh chấp của nhau cùng một lúc và đảm bảo tính nhất quán của dữ liệu chia sẻ.

1. Có những yêu cầu nào dành cho lời giải của bài toán vùng tranh chấp?

* Lời giải cho bài toán vùng tranh chấp phải đảm bảo 03 yêu cầu sau:
  + Mutual exclusion (loại trừ tương hỗ): Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.
  + Progress (tiến triển): Một tiến trình tạm dừng bên ngoài vùng tranh chấp không được ngăn cản các tiến trình khác vào vùng tranh chấp.
  + Bounded waiting (chờ đợi giới hạn): Mỗi process chỉ phải chờ để được vào vùng tranh chấp trong một khoảng thời gian có hạn định nào đó. Không xảy ra tình trạng đói tài nguyên (starvation).

1. Có mấy loại giải pháp đồng bộ? Kể tên và trình bày đặc điểm của các loại giải pháp đó?

* Có 2 nhóm giải pháp chính:
  + Nhóm giải pháp Busy Waiting :
    - Tính chất :
      * Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng (thông qua việc kiểm tra điều kiện vào CS liên tục).
      * Không đòi hỏi sự trợ giúp của hệ điều hành.
  + A close up of a text

    Description automatically generatedCơ chế chung:
  + Bao gồm một vài loại:
    - Sử dụng các biến cờ hiệu.
    - Sử dụng việc kiểm tra luân phiên.
    - Giải pháp của Peterson.
    - Cấm ngắt (giải pháp phần cứng – hardware).
    - Chỉ thị TSL (giải pháp phần cứng – hardware).
  + Nhóm giải pháp Sleep & Wakeup.
    - Tính chất:
      * Từ bỏ CPU khi chưa được vào CS.
      * Cần sự hỗ trợ từ hệ điều hành (để đánh thức process và đưa process vào trạng thái blocked).
  + Cơ chế chung :A close-up of a text

    Description automatically generated
  + Bao gồm một vài loại :
    - Semaphore.
    - Monitor.
    - Message.

1. Phân tích và đánh giá ưu, nhược điểm của các giải pháp đồng bộ busy waiting (cả phần cứng và phần mềm)?

* Các Giải Pháp Phần Cứng:
  + Cấp Ngắt (Interrupts):
    - Ưu điểm:
      * Hiệu suất cao: Cấp ngắt giúp tối ưu hóa hiệu suất bằng cách cho phép hệ thống phản hồi ngay lập tức khi có sự kiện xảy ra, không cần tiến trình phải chờ đợi hoặc kiểm tra liên tục.
      * Tương thích với nhiều loại hệ thống: Có thể sử dụng cấp ngắt trong các hệ thống phức tạp và nhiều loại kiến trúc khác nhau.
    - Nhược điểm:
      * Khó triển khai: Việc lập trình và quản lý cấp ngắt có thể phức tạp và yêu cầu kiến thức sâu về phần cứng.
  + Chỉ Thị Test-and-Set Lock (TSL):
    - Ưu điểm:
      * Đơn giản và hiệu quả: Chỉ thị TSL cung cấp một cơ chế đồng bộ đơn giản và hiệu quả, cho phép kiểm tra và thiết lập một biến trong một lần thực thi.
    - Nhược điểm:
      * Busy waiting: Nếu không được xử lý đúng cách, có thể dẫn đến busy waiting, tiêu tốn tài nguyên CPU khi các tiến trình đợi vào vùng tranh chấp.
* Các Giải Pháp Phần Mềm:
  + Sử Dụng Giải Thuật Kiểm Tra Luân Phiên:
    - Ưu điểm:
      * Đơn giản: Giải pháp này đơn giản và dễ triển khai.
      * Không yêu cầu phần cứng đặc biệt: Không yêu cầu tính năng cụ thể từ phần cứng.
    - Nhược điểm:
      * Busy waiting: Tiêu tốn nhiều thời gian CPU vì các tiến trình phải liên tục kiểm tra trạng thái.
  + Sử Dụng Các Biến Cờ Hiệu:
    - Ưu điểm:
      * Đơn giản: Sử dụng biến cờ hiệu để đánh dấu trạng thái của vùng tranh chấp.
    - Nhược điểm:
      * Busy waiting: Cũng có thể dẫn đến busy waiting và tiêu tốn nhiều tài nguyên CPU.
  + Giải Pháp Của Peterson:
    - Ưu điểm:
      * Hiệu quả với số lượng nhỏ tiến trình: Hiệu quả khi chỉ có hai tiến trình tham gia vùng tranh chấp.
    - Nhược điểm:
      * Busy waiting: Có thể gây ra busy waiting khi số lượng tiến trình tăng.
  + Giải Pháp Bakery:
    - Ưu điểm:
      * Quản lý tốt thứ tự truy cập: Hiệu quả trong việc quản lý thứ tự truy cập vào vùng tranh chấp.
    - Nhược điểm:
      * Busy waiting: Có thể dẫn đến busy waiting, không phù hợp với số lượng tiến trình lớn.

1. Semaphore là gì? Đặc điểm của semaphore? Cách thức hiện thực semaphore? Có mấy loại semaphore? Khi sử dụng semaphore cần lưu ý những vấn đề gì?

* Semaphore là một cơ chế đồng bộ hóa được cung cấp bởi hệ điều hành để quản lý việc truy cập vào tài nguyên được chia sẻ. Đây là một biến số nguyên có cấu trúc đặc biệt, đi kèm với hai hoặc nhiều hơn các hoạt động cơ bản: wait (P) và signal (V).
* Đặc Điểm Của Semaphore:
  + Hoạt Động Giữa Các Tiến Trình: Semaphore quản lý quyền truy cập vào tài nguyên chia sẻ giữa các tiến trình hoặc luồng trong môi trường đa nhiệm.
  + Hành Vi wait (P): Giảm giá trị của semaphore (S = S - 1). Nếu giá trị mới của semaphore âm, tiến trình gọi wait sẽ bị chặn (blocked).
  + Hành Vi signal (V): Tăng giá trị của semaphore (S = S + 1). Nếu giá trị mới của semaphore không dương, một tiến trình đang bị chặn bởi wait sẽ được kích hoạt.
* Cách Thức Hiện Thực Semaphore:
  + Semaphore có thể được thực hiện bằng cách sử dụng một cấu trúc dữ liệu có chứa một biến nguyên (đại diện cho giá trị của semaphore) và một hàng đợi (queue) chứa các tiến trình hoặc luồng đang chờ đợi sự kiện của semaphore.
* Loại Semaphore:
  + Binary Semaphore (Semaphore Nhị Phân): Có giá trị 0 hoặc 1, thường được sử dụng để đồng bộ hóa giữa hai tiến trình.
  + Counting Semaphore (Semaphore Đếm): Có giá trị lớn hơn 0, cho phép nhiều tiến trình truy cập vào một tài nguyên.
* Lưu Ý Khi Sử Dụng Semaphore:
  + Busy Waiting: Tránh sử dụng busy waiting (chờ đợi bằng cách liên tục kiểm tra) để tiết kiệm tài nguyên CPU.
  + Đồng Bộ Hóa Đúng Đắn: Semaphore cần được sử dụng một cách đồng bộ và chính xác để tránh deadlock hoặc starvation.
  + Atomic và Mutual Exclusion: Cần đảm bảo rằng các hoạt động trên semaphore là atomic và đảm bảo mutual exclusion giữa các tiến trình.

1. Monitor và Critical Region là gì?

* Critical Region:
  + Critical Region (CR): Là một phần của mã chương trình mà khi một tiến trình đang thực thi trong nó, không cho phép các tiến trình khác thực hiện cùng một đoạn mã đó đồng thời.
  + Đặc Điểm: Cần đảm bảo chỉ một tiến trình được phép thực thi trong critical region tại một thời điểm để tránh race condition và bảo đảm tính nhất quán của dữ liệu.
* Monitor:
  + Monitor: Là một kiểu dữ liệu trừu tượng trong lập trình đa luồng, dùng để quản lý truy cập vào các biến chia sẻ và các phương thức tương tác với chúng.
  + Đặc Điểm: Chỉ các phương thức được định nghĩa trong monitor mới có thể truy cập và thao tác trên các biến chia sẻ trong monitor đó.
  + Chức Năng Tương Tự Semaphore: Monitor có chức năng tương tự như Semaphore trong việc đồng bộ hóa và quản lý truy cập tài nguyên chia sẻ, nhưng cung cấp cách tiếp cận dễ dàng hơn và đảm bảo tính chất an toàn hơn.

1. Đặc điểm và yêu cầu đồng bộ của các bài toán đồng bộ kinh điển?

* Đặc Điểm:
  + Tài Nguyên Chia Sẻ: Tất cả các bài toán đồng bộ kinh điển đều liên quan đến việc quản lý truy cập vào tài nguyên chung (dữ liệu, thiết bị, vùng nhớ, v.v.) giữa các tiến trình hoặc luồng.
  + Race Condition: Việc không đồng bộ hóa truy cập vào tài nguyên chia sẻ có thể dẫn đến race condition, khi nhiều tiến trình cùng thao tác và ghi đọc dữ liệu cùng một lúc, dẫn đến kết quả không đúng.
* Yêu Cầu Đồng Bộ:
  + Mutual Exclusion (Khóa Độc Quyền): Một tiến trình khi đang thực hiện trong vùng critical section thì không được phép có tiến trình nào khác cùng thực hiện trong vùng đó.
  + Progress (Tiến Triển): Nếu không có tiến trình nào ở trong vùng critical section và có một số tiến trình muốn vào, chỉ có tiến trình không phải là ở trong vùng critical section mới được phép vào.
  + Bounded Waiting (Hạn Chế Chờ Đợi): Cần đảm bảo rằng một tiến trình không phải chờ đợi vô hạn lâu để vào vùng critical section. Nghĩa là sau một thời gian nhất định, mọi tiến trình muốn vào vùng critical section sẽ có cơ hội vào.
* Các Bài Toán Đồng Bộ Kinh Điển:
  + Bài Toán Tiền Lương (The Dining Philosophers Problem): Nhiều triết gia ngồi xung quanh một bàn tròn, mỗi người cần sử dụng đũa bên trái và bên phải của mình, nhưng không được phép cùng một lúc sử dụng đồ ăn từ cả hai bên.
  + Bài Toán Producer-Consumer Problem: Một producer sản xuất các sản phẩm để đặt vào một vùng chứa, trong khi consumer loại bỏ các sản phẩm từ vùng chứa đó. Cần đồng bộ hóa để tránh tình trạng dữ liệu bị mất hoặc tràn bộ nhớ.
  + Bài Toán Đọc-Ghi CSDL (Readers-Writers Problem): Nhiều người đọc có thể đọc từ một tài nguyên cùng một lúc, nhưng chỉ có một người viết mới được phép ghi vào tài nguyên đó và không được phép đọc.

1. (Bài tập mẫu)

* Ta thấy code này có đoạn j = 1 – i => đề chỉ đang nói tới xét 2 tiê trình P1 và P0 vì khi i = 0 thì j = 1 và ngược lại.
* Giải thuật này không thoả mãn: Xét tình huống khi flag[0] =1; turn =0; lúc này P0 vào CS, Nếu lúc đó flag[1] = 1, P1 có thể gán turn = 1 và vào luôn CS (2 tiến trình cùng vào CS một lúc).
* Có thể sử dụng chỉ thị Swap để tổ chức truy xuất độc quyền

|  |
| --- |
| var      lock: boolean;  procedure Swap(var a, b: boolean);  var      temp: boolean;  begin      temp := a;      a := b;      b := temp;  end;  procedure EnterCritical(var lock: boolean);  var      key: boolean;  begin      key := true;      while key do      begin          Swap(lock, key);      end;  end;  procedure LeaveCritical(var lock: boolean);  begin      lock := false;  end;  begin      lock := false;      EnterCritical(lock);      // critical section      LeaveCritical(lock);      // remainder section  end. |

* a)

|  |
| --- |
| Semaphore semaphoreA = 0; // Semaphore cho tiến trình A  Semaphore semaphoreB = 9; // Semaphore cho tiến trình B  Process A {  while (TRUE) {  wait(semaphoreA);  na = na + 1;  signal(semaphoreB);  }  }  Process B {  while (TRUE) {  wait(semaphoreB);  nb = nb + 1;  signal(semaphoreA);  }  } |

* b)

|  |
| --- |
| Semaphore semaphoreB = 10; // Semaphore cho tiến trình B  Process A {  while (TRUE) {  wait(semaphoreB);  na = na + 1;  }  }  Process B {  while (TRUE) {  nb = nb + 1;  signal(semaphoreB);  }  } |

* c) Đúng, vì có thể có nhiều tiến trình loại A hoặc loại B cùng thực hiện nhưng chỉ có 2 biến Semaphore toàn cục mà chúng sẽ thao tác (đối với câu b là 1 Semaphore).

1. (Bài tập mẫu)

|  |
| --- |
| Biến X có thể vượt quá 20 do hai tiến trình cùng thực hiện đoạn code một cách song song. Điều này có thể xảy ra khi một tiến trình đang thực hiện lệnh X = X + 1; và trước khi nó kiểm tra if (X == 20) X = 0;, tiến trình kia đã tăng X lên 1, khiến X vượt quá 20.  Để khắc phục vấn đề này, bạn cần sử dụng một cơ chế đồng bộ hóa để đảm bảo rằng chỉ có một tiến trình được thực hiện đoạn code này tại một thời điểm. Một cách để làm điều này là sử dụng semaphore:  Semaphore mutex = 1; // Khởi tạo semaphore  do {  wait(mutex); // Đợi cho đến khi có quyền truy cập vào biến X  X = X + 1;  if (X == 20) X = 0;  signal(mutex); // Cho phép tiến trình khác truy cập vào biến X  } while (TRUE);  Trong đoạn mã trên, semaphore mutex được sử dụng để kiểm soát quyền truy cập vào biến X. Khi một tiến trình muốn thực hiện đoạn code, nó sẽ gọi wait(mutex) để đợi cho đến khi có quyền truy cập. Khi nó hoàn thành, nó sẽ gọi signal(mutex) để cho phép tiến trình khác truy cập vào biến X. Điều này đảm bảo rằng chỉ có một tiến trình được thực hiện đoạn code này tại một thời điểm, ngăn chặn X vượt quá 20. |

|  |
| --- |
| Semaphore semaphore1 = 0;  Semaphore semaphore2 = 0;  Process P1 {  A1;  wait(semaphore1);  signal(semaphore2);  A2;  }  Process P2 {  B1;  wait(semaphore2);  signal(semaphore1);  B2;  } |

|  |
| --- |
| Semaphore\_1= 1;  Semaphore\_2= 1;  Process A:  for (int i=1; i<100; i++)  {  wait(Semaphore\_1);  Ai;  signal(Semaphore\_2);  }  Process B:  for (int i=1; i<100; i++)  {  wait(Semaphore\_2);  Bi;  signal(Semaphore\_1);  } |

|  |
| --- |
| // Process 1  w = x1 \* x2;  signal (s15);  signal (s16);  // Process 2  v = x3 \* x4;  signal(s23);  signal(s24);  // Process 3  wait(s23);  y = v \* x5;  signal(s35);  // Process 4  wait(s24);  z = v \* x6;  signal(s46);  // Process 5  wait(s15);  wait(s35);  y = w \* y;  signal(s57);  // Process 6  wait(s16);  wait(s46);  z = w \* z;  signal(s67);  // Process 7  wait(s57);  wait(s67);  ans = y + z; |